DTIC

Software Infrastructure to Enable Parallel Spatial Data Handling

First, Second, Third Interim

and

Final Technical Report

by

M.J.Mineter, S.Dowers, B.M.Gittings

(June 2001)

United States Army

EUROPEAN RESEARCH OFFICE OF THE U.S. ARMY

London, England

R&D 8707-EN-01 Contract N68171 00 M 5807

University Of Edinburgh

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	June, 2001	Interim & Final: 6-JUN-00:5-JUN-01

4. TITLE AND SUBTITLE

Software Infrastructure to Enable Parallel Spatial Data Handling

5. FUNDING NUMBERS

C N68171-00-M-5807

6. AUTHOR(S)

M.J.Mineter, S.Dowers and B.M.Gittings

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

The University of Edinburgh
Department of Geography
Old College, South Bridge
Edinburgh EH8 9LY.

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

USARDSG-UK
Edison House
223 Old Marylebone Road,
London NW1 5TH

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The report describes research intended to enhance the timeliness and scope for intervisibility analyses to improve decisions related to intervisibility. The research entailed the development and use of prototype software to:

1. Manage the completion of a large number of visibility analyses, each of which determines the visible regions from a point in a digital elevation model. The project accomplishes the visibility analyses by efficient use of spare CPU cycles on a network of processors.
2. Build a database from these analyses. The database is sufficiently compact for a test grid of 466 rows, each with 336 columns to reside comfortably on a CD (it requires 84Mbytes).
3. Extract information from the database using a series of tactical decision aids.

The concurrent execution of multiple visibility commands, accomplished by the development of a multicomputing framework achieved significant speed-up, limited only by the number of processors available (13). The compression methods adopted for the database achieved ~97% reduction in volume, whilst still allowing timely access to the data to support interactive use of tactical decision aids.

14. SUBJECT TERMS

Computing; Software; Spatial Data; GIS; Parallel processing; multi-computing; performance; intervisibility

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Abstract

The report describes research intended to enhance the timeliness and scope for intervisibility analyses to improve decisions related to intervisibility. The research entailed the development and use of prototype software to:

1. Manage the completion of a large number of visibility analyses, each of which determines the visible regions from a point in a digital elevation model. The project accomplishes the visibility analyses by efficient use of spare CPU cycles on a network of processors.
2. Build a database from these analyses. The database is sufficiently compact for a test grid of 466 rows, each with 336 columns to reside comfortably on a CD (it requires 84Mbytes).
3. Extract information from the database using a series of tactical decision aids.

The concurrent execution of multiple visibility commands, accomplished by the development of a multicomputing framework achieved significant speed-up, limited only by the number of processors available (13). The compression methods adopted for the database achieved ~97% reduction in volume, whilst still allowing timely access to the data to support interactive use of tactical decision aids.

1 Overview of body of the report

This document reports on a 3-month project undertaken by the Department of Geography in the University of Edinburgh, to explore techniques for enhancing the computer-based analysis of terrain to aid in decisions related to intervisibility.

The goal of the project was to investigate and prototype multicomputing software, to construct a database of intervisibility data, and to develop several demonstration tactical decision aids. The concurrency is achieved by executing ARCINFO upon a number of networked processors, utilising spare processing time.

Following a restatement of the project specification, the report describes each of the major components of the project in turn: the multicomputing architecture, the building of the database and the tactical decision aids. In each case the discussion includes consideration of the usability of the component, and the scope for productisation and further development.

Appendices give further technical information, focussing upon orientation for those who seek to run or further develop the prototypes. An associated CD includes this Final Report, a power-point presentation, all software (with documentation of the Java tactical aids), and a complete intervisibility database for the test dataset.

2 Project specification

2.1 Background

Intervisibility is the term used to describe the effects of terrain on visibility. It is a key factor in military terrain analysis and impacts a soldier's field-of-view, viewing distance, and engagement ranges. A number of different intervisibility products are available. Point-to-point intervisibility results in a line-of-sight profile (Figure 1a). Single point to multiple point intervisibility produces a masked area plot (Figure 1b). In both figures, green indicates visibility and red indicates no visibility. Intervisibility products have been largely limited to line-of-sight profiles and masked area plots because of the amount of time required to generate individual products.

Clark Ray and Bob Richbourg proposed a new intervisibility product, the complete intervisibility plot. This new product stored information on the cumulative intervisibility from every post to every other post in a Digital Elevation Model (DEM). This product is useful because it can provide information on the relative visibility of locations in a DEM. While a complete intervisibility plot opens up a new area of terrain analysis research, it is difficult and time-consuming to compute. Ray developed an approximation technique to speed the calculations, because it was thought that it would be too time-consuming to generate the product using standard processing techniques. In performing the calculations, Ray did not maintain the intermediate intervisibility plots, but rather stored the cumulative relative intervisibility.

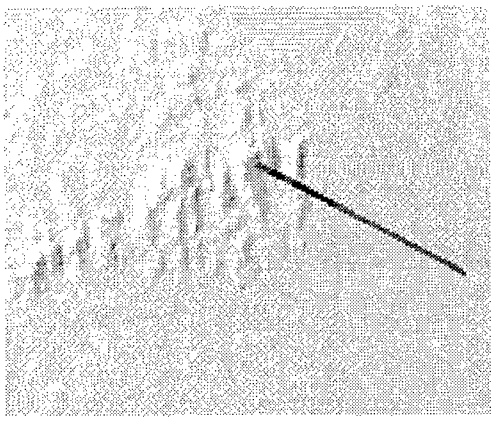


Figure 1a : Line of Sight Profile

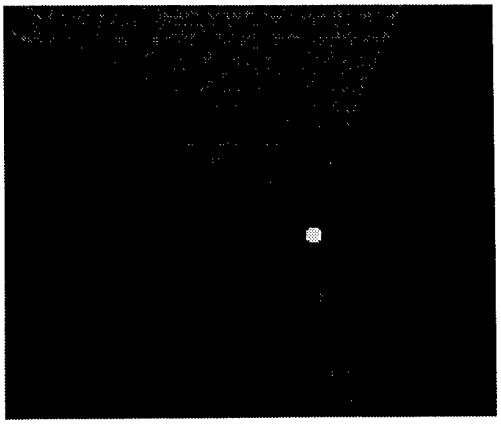


Figure 1b : Masked Area Plot

This effort will attempt to re-examine the generation of complete intervisibility information within a multicomputing environment. A Complete Intervisibility Database (CID) is a database consisting of the masked area plot information for all posts in an elevation model. For a small-sized 400 x 500 DEM, this would consist of 200,000 masked area plots. Unlike Ray's work, intermediate results will be maintained so that they can be used to generate multiple Tactical Decision Aids.

While it may be too time-consuming to calculate complete intervisibility on a single machine, it should not be difficult to break the job into smaller tasks and complete them on multiple machines. This will be done by scheduling the jobs to run at low priority, when the computing power is under-utilized.

This effort will develop new and innovative techniques to create, store and exploit a Complete Intervisibility Databases. In addition, it will stimulate additional research and analysis related to a intervisibility analysis. This information can be used to investigate the relationships between intervisibility and terrain, to evaluate issues associated with Ray's approximation technique, as well as to develop a new generation of tactical decision aids based on the Complete Intervisibility Database.

2.2 Goals

The four goals of this effort are to...

1) Develop and implement a multicomputing architecture for generating and exploiting a Complete Intervisibility Database.

2) Design a data structure to store the results of a Complete Intervisibility Database

3) Generate a Complete Intervisibility Database for a 1:24,000-scale U.S. Geological Survey DEM

4) Develop three (3) new tactical decision aids based on the Complete Intervisibility Database

2.3 Tasks

2.3.1 Task #1: Multicomputing Architecture for the Complete Intervisibility Database

The first task was to develop the multicomputing architecture for the Complete Intervisibility Database. The proposed configuration would utilize multiple UNIX computers running ArcInfo on a network. ArcInfo would be used for the visibility analysis and data manipulation. The architecture would provide for the following Production Management capabilities

a. Register available computers with appropriate licenses on the network

b. Manage the division of the tasks

c. Determine computer availability for processing

c. Allocate of tasks to multiple computers

d. Monitor the status of on-going jobs

e. Verify that jobs have been completed successfully

f. Report on status of work, i.e., jobs completed, jobs in progress, remaining jobs

2.3.2 Task #2: Data Structure and Format for Storing Complete Intervisibility Database

The Complete Intervisibility Database represents a new concept in terrain analysis and the optimal storage configuration for the results has yet to be designed. It would be possible to store the results as a series of ArcInfo grids, but the management and overhead associated with this approach may be unreasonable. For example, for a 1201 x 1201 elevation matrix, there would be 1,442,401 grids.

The proposed structure for storing the Complete Intervisibility Database would facilitate rapid access and analysis of the datain a multicomputing environmentAccess was emphasized over minimization of the storage requirements.

2.3.3 Task #3: Complete Intervisibility Database

In order to demonstrate the working multi-computer architecture, the University of Edinburgh was to create a Complete Intervisibility Database for a U.S. Geological Survey 1:24,000-scale Digital Elevation Model (DEM). The source data set was a relatively small DEM with 466 rows and 336 columns. There were 156576 masked area plots in the Complete Intervisibility Database. This compares with the National Imagery and Mapping Agency's standard Digital Terrain Elevation Data (DTED) Level 1 data set with 1201 rows and 1201 columns and the DTED Level 2 data set with 3601 rows and 3601 columns.

The resulting Complete Intervisibility Database would be generated using the multicomputing architecture specified in Task #1 and the results will be stored in the data structure and format specified in Task #2.

2.3.4 Task #4: Complete Intervisibility Database Tactical Decision Aids

The Complete Intervisibility Database would be used with exploitation software to generate at least three (3) tactical decision aids. These would include at a minimum

a. Cumulative Visibility. For each post in the DEM, the cumulative visibility would be calculated. This would be determined by counting the number of masked area plots in which the post is visible.

b. Post of Maximum Visibility. For each post in the DEM, the post of maximum visibility would be calculated. The resulting table would store the location of the masked area plot or plots that covers the post and has the maximum number of visible posts. c. Multi-Observer Masked Area Plot. For a user-specified polygon, a multi-observer masked area plot would be calculated. This would be determined by summing the masked area plots for all locations for multiple observation points. Posts with high values will be locations that are most visible.

3 Project achievements

1) The multicomputing architecture has been completed and tested across 13 UNIX processors. We used a 'task farm' approach: each workstation requests a new task when it is ready to process it. One processor also acts as coordinator of the task farm. The coordination is achieved using PERL, remote shells, and shared directories. The tasks are performed using local disk space, results being copied to a final directory on a shared disk.

   It includes the ability to:

   a) Manage which processors are active in any run.

   b) Distribute visibility analyses to each processor.

   c) Respond to computer availability. By running process at low priority, spare CPU time is used. The ability for a workstation to assess its spare capacity before requesting a task is not implemented: the hooks for this exist.

   d) Monitor the status of on-going jobs. A monitor utility, run by a user:

      i) Lists jobs in progress.

      ii) Lists jobs that have failed (identified by generation by the job of an ARC error message.) These jobs will be rerun when the task farm is next started, as errors might be due to circumstances needing user intervention.

      iii) Generates statistics concerning use of CPU time

      iv) Recognises failed processors based upon the average CPU time per job and the elapsed time since the last job was started. This utility can cause the failed processor to be restarted within the task farm.

   e) Write statistics to disk periodically, to profile CPU use and task completion across the architecture.

   f) Re-attach a failed processor to the task farm.

   g) Close the task farm at a specified time, holding the status of completed jobs for the next occasion when the task farm is used. (This allows scheduling of the task farm if desired, for example from 7pm to 7am daily, and over weekends. An alternative mode of use is offered by the running of tasks at low priority: in many cases the task farm will be started then run to completion.)

   h) Restart the task farm at a specified time, filling gaps in any missed or previously failed tasks, then continuing from the run reached previously.

2) Data structure and Format for storing the Complete Intervisibility Database

This has been designed, implemented and tested. It is described below. A simple compressed representation of several visibility grids is used; many such files exist indexed by directory names.

3) Complete Intervisibility Database (CID)

This has been generated.

4) Tactical Decision Aids. These have been coded and tested..

In addition:

5) A Java utility has been coded to allow interrogation of the database and of Decision Aid results.

6) A utility has been written to extract one view

7) A utility takes an ArcInfo DTM file and installs it for use by the multiprocessing architecture.

8) Several minor scripts:

   a) List processes owned by the user on all active processors.

   b) Check for the first incomplete process.

4 The Multicomputing Architecture

4.1 Overview

The goal of supporting the multiple visibility analyses within ArcInfo was implemented as a prototype software system. This was done as a generic architecture to support a class of problems in multicomputing, a subset of what is sometimes termed high-throughput computing. The functionality is as follows:

1) Utilise multiple processors to run a large number of tasks of typical run-time of a few minutes. Due to the relatively short run-times, check-pointing and migration of processes between processors is not required. For example, using the test dataset provided, 9828 tasks were required, with run times of between 2 and 5 minutes, depending on which processor was used.

2) Minimise management of worker processors by an operator.

3) To allow coordinator and workers to be run/stopped/powered off at any time and the coordination of tasks to be recoverable.

4) The mechanism used is to be generic, to support visibility analyses or other similar concurrent tasking, using GIS or not, where a simple script can cause the task to run.

5) Use standard operating systems and functions. This ruled out MOSIX, which supports multi-processing on LINUX clusters.

This functionality is achieved by parallelism at a level above that of the individual processes: speeding the execution of one visibility analysis by using multiple processors would be impractical in 3 months, and would in any case be less efficient than running multiple processors each on one visibility analysis. In general parallelism is most efficient when applied at the highest practical level. Given that more than 9,828 analyses are required for the test dataset, until more than this number of processors is available no benefit is derived from the low

level parallelism. Furthermore, many larger datasets, with correspondingly more tasks, are likely to be used.

The project task was accomplished by:

1) A review of pre-existing multicomputing software. Most related multi-processing frameworks are more complex than is needed here. Nothing suitable was found. However, noteworthy is CONDOR, http://www.cs.wisc.edu/condor, which supports processing where each task runs much longer than is required here... it allows check-pointing and migration of processes, for example.

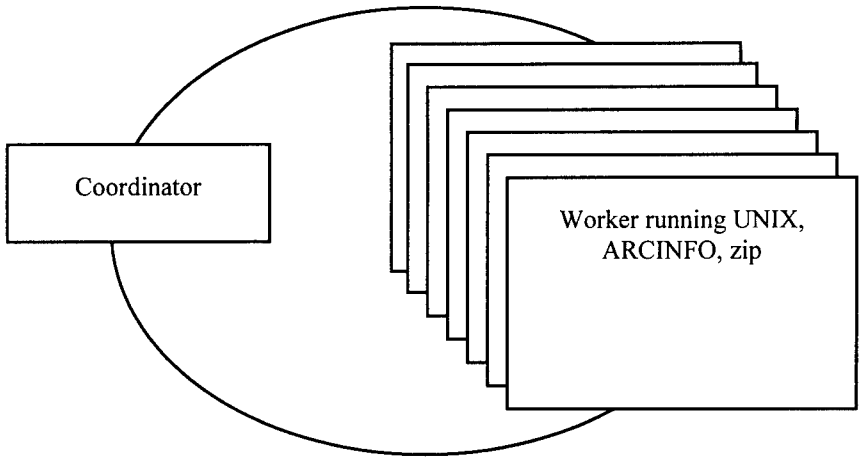
2) A review of possible methodologies. TCP/IP messaging incurs minimum overhead, but results in more complex process management. Java supports remote method invocation, but this was considered over-sophisticated, and again less transparent to the user and maintainer of the architecture. Object-oriented methods using request brokers and many servers were also considered over complex for this duration of project. A simpler method, more transparent to the operator of the system, was tested and found more than adequate.

3) Implementing an architecture based on a combination of remote shells invoked by one coordinating process and use of one shared filespace, from and to which each worker process read/wrote data.

4) Developing utilities for profiling and monitoring use.

4.2 Design

The Architecture comprises a Coordinator process and multiple Workers (each comprising a different processor).



Multiple processes, each residing on a different processor on a local area network

Figure 2 : Task farm: coordinator process and multiple worker processes

The coordinator receives requests for a task (in this case, a visibility analysis) from each worker, when that worker has either initialised or else completed the previous task. Such an architecture is termed a task farm, and is commonly used for low level parallelism (for example to process different sub-sets of data in a parallel program). As described further below, in this case a task comprises the execution of a ArcInfo command, and the zip compression of results. Worker processors were running UNIX in this prototype.

A run is controlled as follows (Figures 3-5):

1 A *controlTaskFarm* script is run on the coordinating machine. This is the only user action required. The script will download to each worker, by *rcp*, and run by *rsh*, a *workerInit* script on each worker.

2 The *workerInit* script copies files from shared directories to the workers own workspace, to be ready to run tasks, and sends a request for a task back to the coordinator. The consequence of this is that updates to scripts are propagated to all workers at initialisation: setup of workers is automatic. The request comprises writing a file with the worker's name encoded within it, to a shared directory that is periodically (every 5sec) checked by the coordinator. The script could be extended to wait until the processor activity falls below a threshold. This has not been coded: it was simply preferred to run all tasks at a low priority.

3 On recognition that a worker requires a task (by receipt of data/message/file written by that worker) the coordinator determines the next task to be carried out, and invokes this on the specified worker by *rsh* to run the *workerTask* script on the Worker.

4 The *workerTask* script causes a task to be carried out, and copies the results to the shared directories of the database. (In some tasks a Sink process, to collate these data would be needed: this is not necessary here, as once written to the shared directory no further processing is required to build a database.) The *workerTask* script then requests a new task from the coordinator, using the same mechanism as above. It also generates a trace file for each task, and scans this for errors. If one is found, then the trace file is copied to the coordinator's *taskFailed* directory.

Processing thus proceeds with the coordinator allocating tasks as workers become available, so the changing availabilities of processors (due to other computation) or the different speeds of processors is managed within the task farm: there is no pre-allocation of particular tasks to particular workers.

Closedown happens when all tasks have been run, or when an operator intervenes by runnning the script *causeStop*, which could be invoked (as is the case for *controlTaskFarm,* from a timed batch queue). The task farm then ceases sending more tasks to workers, and waits until the current tasks are completed.

The use of disk files, rather than something more elegant like TCP/IP, allows access to the state-of-play by other scripts and also by an operator, in particular to monitor the tasks being executed. It might also better suit the focus upon not running daemons on workers, but ensuring that if the coordinating process is closed then all workers are likely also to close.

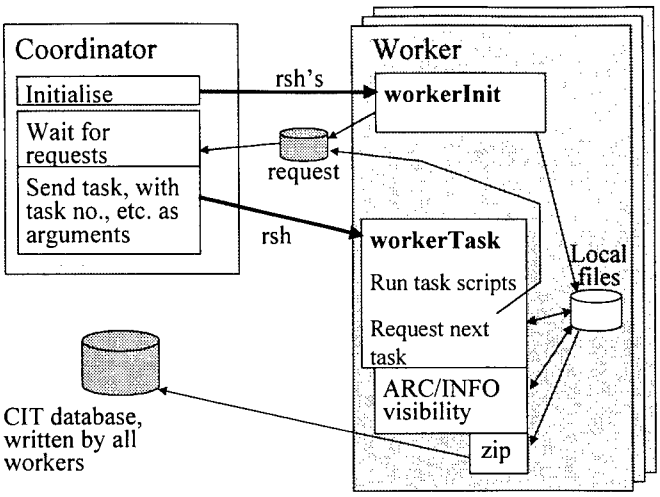


Figure 3 : Overview of task farm

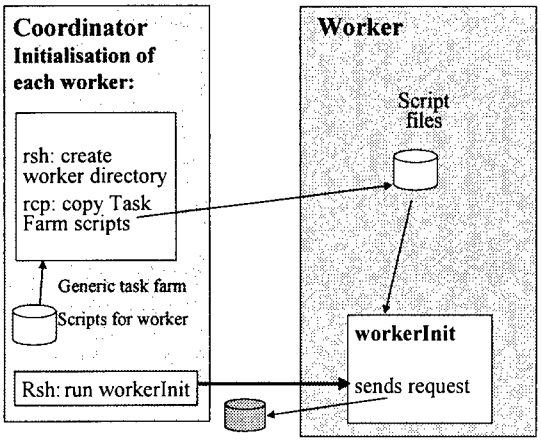


Figure 4 : Initialisation of worker

Mechanisms used are:

- PERL
- Shared directories as a means of communication from workers to the coordinator:
  - To send requests
  - To write results.
- Rsh is used by the coordinator to invoke remote shells on workers. Using ssh would be preferable in "live" use of the software, due to security issues. Each rsh comand is invoked from a forked process from within the coordinator. Consequently, no daemons are run on worker machines, and abrupt closedown can be achieved from the coordinator, if required.

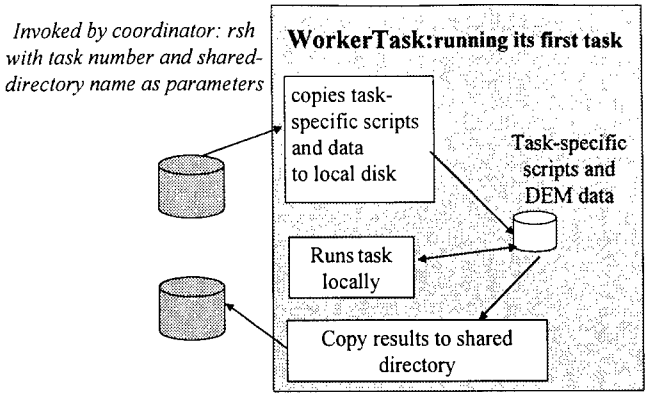


Figure 5 : Worker processing to run a task

4.3 *Performance*

The overheads of the architecture software did not significantly limit performance in the tests run in Edinburgh: the ArcInfo visibility analysis was the dominant user of CPU time, as would be expected. Performance figures gathered in the context of the visibility analyses to build the database are discussed in the following section.

One observation indicated that the overheads of the task farm were small. The visibility analysis scripts check to see if the data already exist for a particular task; if so then a task is not repeated. The worker carries out this check. This characteristic was used to assess the speed with which tasks were assigned by the architecture. The run was set up to repeat 326 tasks, for which data already existed. Using 11 processors, these tasks were allocated in 3mins 6secs, during a term-time afternoon. This was with the task farm configured to wait for 5 seconds between polling for requests (chosen as the default, and used in other tests). Reducing this period to 2secs reduced the run-time for this test to 2min 32secs. Other values were not tried. For tasks of very short duration, and/or for many workers, efficiency could be further increased if one task-farm task caused several application tasks to run.

4.4 *Possible developments*

The prototype was also intended to allow, with some limited further development:

1) different types of task to be performed either concurrently or consecutively

2) determination of what processing was required by a task during a run. The goal was to allow future work to support applications in which the results of one series of runs influenced the commands executed by future tasks. In the present software, this is not implemented, but is considered to be a viable development.

For more than use as a prototype some tidy-up of code would be desirable:

1 Some additional diagnosis of error conditions would be required:

2 At present there are some remnants of first-version mechanisms in the code (checking for completion by the existence of a ".done" file rather than by checks of the CID database itself. The latter is now done after a run to set up the finished.tasks file with the number of the task before a gap in the database.).

3 The use of finished.tasks is imprecise after the last task has been run; the use of total.tasks (a file set up by the installdata script) may best be superceded by a different mechanism to recognise the last task to be run. It would be preferable to allow the last task to be determined during the run, in some cases perhaps. This approach is acceptable for generation of the CID.

4 The monitor utility does not check if the last allocated task for a processor has yet completed.

5 The decoupling of the task-specific scripts and the task farm control was compromised to check the database after a run. A cleaner interface between the task and the task farm is desirable.

6 The shared directories are assumed to have the same pathname on all machines. This is ok for the configuration in Edinburgh, but might be a constraint in general.

7 The controlTaskFarm, monitor, and causeStop scripts, together with the various other minor scripts useful to snap-shot the database and system activity could be better packaged to aid future use.

5 Building the Complete Intervisibility Database

5.1 Overview

The database is required to comprise the visibility analysis from each grid point, held in such a way that the specified range of decision tools can be executed in a reasonably timely manner. Given the timescales for the project it was decided to maximise the exploitation of existing tools, and seek a simple database structure, one which could in future be distributed across multiple disks to facilitate parallel implementations of the decision tools. The approach was as follows:

1 Use ArcInfo to process 16 points at a time.

2 Use zip compression on each of these files.

3 Use simple directory names to index these files by row.

4 Use the capabilities of Java in reading the zipped data.

The ability of ArcInfo (the required tool for the visibility analysis itself) to run up to 16 analyses at once allowed a sub-grid of 4 by 4 points to be processed in each command. This was thought likely to maximise the similarity between the results from each point, and hence improve the scope for data compression.

5.2 Database design

Input is a regular square grid containing elevations at each a post in the centre of each grid element. There are nCols x nRows posts in the grid. Rows are numbered from the top of the grid in the normal image-processing convention.

A Masked Area Plot for the post at column i and row j, MAP(i,j) contains a binary value for each post in the grid indicating whether that post is visible from the originally selected post. The MAP therefore contains nCols x nRows binary values and the simplest representation is an nCols x nRows array of bits.

A Complete Intervisibility Database (CID) is a collection of MAP(i,j) for each post in the original grid. There are therefore nCols x nRows MAPs in the CID and the total size is nCols x nRows x nCols x nRows bits. For an input grid of 336 columns by 466 rows the CID is 24,516,043,776 bits or 2.9Gb. The size of this suggests that some form of compression is desirable.

ArcInfo can process up to 16 observer points in one command, the result being stored in an integer grid with 16-bit values where each bit represents an observer. By selecting the 16 observers in a 4x4 subgrid of the original grid we can expect there to be significant correlation between the bits in a 16-bit word, which will aid in compression. The post processing of a MAP can also be simplified with this form of representation since each 16-bit value corresponds to a post and therefore post-by-post processing is simplified compared with unpacking single bit values. The collection of 16 MAPs together is called a MAP16 and is identified by the column and row of the top-left post. There are therefore (nCols/4) x (nRows/4) MAP16s in the CID. For a 336 by 466 input grid there are 9828 MAP16s. The MAP16s are generated in binary form by converting from ArcInfo grid to BIL format. This database structure has the advantage of simple random access to any MAP16 and expansion is achieved using standard tools available on almost all computing platforms.

Rather than develop specialised compression techniques, we decided to investigate commonly used compression software. Initial tests with very sparse MAPs showed compression factors of between 100 and 300. Based on this we decided to use a very simple database structure based on directories in the native file system. Each row of MAP16s is allocated a separate directory named from the row number. Each directory contains a zip file for each MAP16 in the row with the name based on the column number of the MAP16. The zip file contains a single compressed BIL representation of the MAP16. The top directory of this structure also contains a header file describing the original DEM, projection information and a missing value mask.

example -> header (where -> denotes contains, so directory example contains header, proj, mask etc)

```
proj
dtm  (an ArcInfo ASCII grid of the heights)
mask (used for missing values)
0 ->  0_im.zip ->     0_im.bil
      4_im.zip ->     4_im.bil
      8_im.zip ->     8_im.bil
      ..
4 ->  0_im.zip -> 0_im.bil
      4_im.zip -> 4_im.bil
      8_im.zip -> 8_im.bil
      ..
8 ->  0_im.zip -
```

Figure 6 illustrates the organisation of the complete database, showing the decomposition of the 4-dimensional structure into a stack of 2-dimensional slices grouped by row.

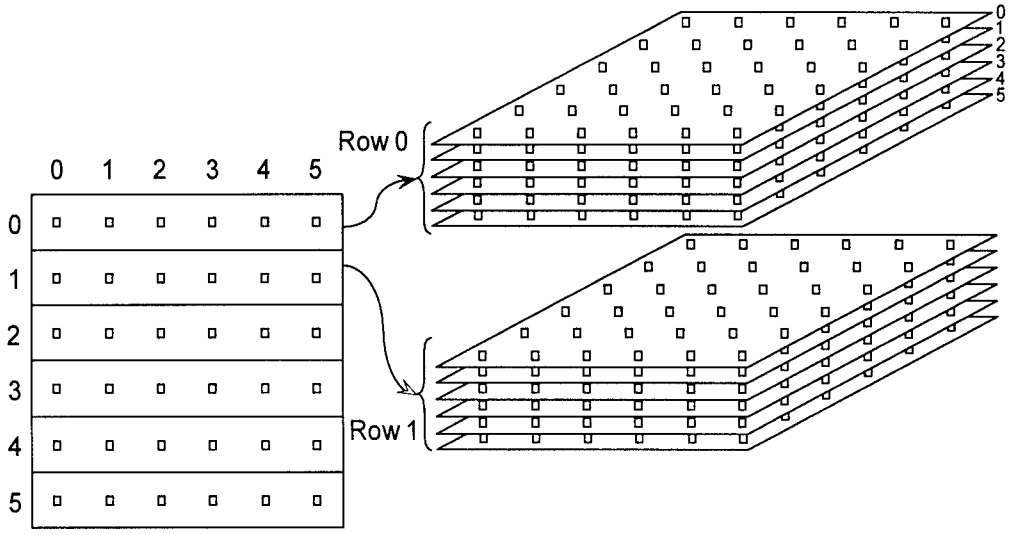


Figure 6 : Complete Intervisibility Database representation

5.3 *Database construction: size and performance*

The final database, with each file separately compressed, is 84Mbytes i.e. approx 3% of the raw 4-D bitmap size (2.9Gb). It comprises 9828 files, with 84 files in each of 117 directories (one directory per row). The distribution of file sizes is shown in figure 7 and spatially in Figure 9, and the disk space used for each row (the sum of the 84 files) is shown in figure 8. The relationship between file size and visibility data has not been explored other than superficially.

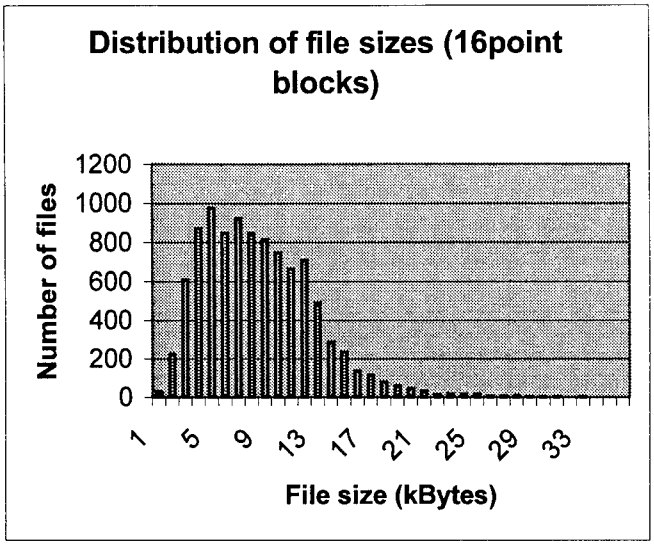


Figure 7 : Distribution of size of compressed MAP16 files

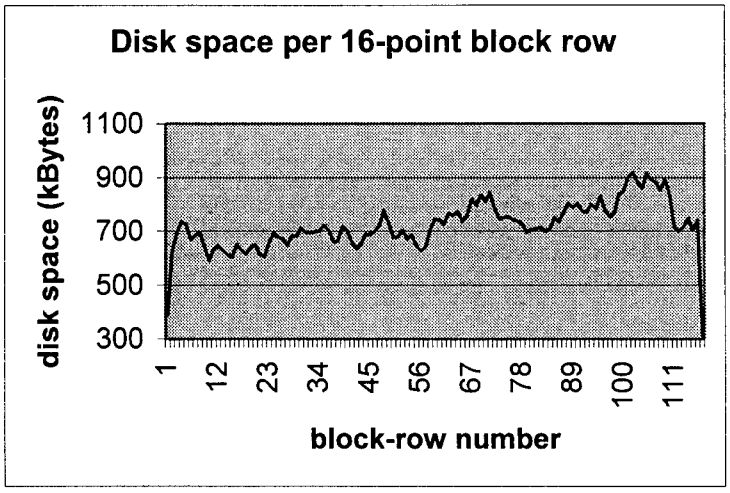


Figure 8 : Total of 84 compressed MAP16 files in each row

On a SUN Sparc 300MHz processor, most runs took between 117 and 130 seconds. A single run performs the visibility analyses for a block of 16 points. In a typical run of 124 seconds, 5 seconds extracted the points for which that analysis was run, and 117secs were used for the visibility analysis.

The complete database was built in 43hrs 26mins using 13 processors, of which 5 were 300MHz Sparcs, the remainder being older Digital processors. Each processor used in excess of 90% of its CPU time in running the tasks. Table 1 is based on an extract the monitor report run on completion of the last task and shows the relative powers of the processors (ts1,2,3,4 represent four workers running on the four-processor Sun server.):

Figure 9 : Variation in size of compressed MAP16 files

```
Total CPU time used on 9828 tasks: 1941929 (=elapsed time*12.42)
 Batting averages:      CPU total  ntasks CPU/task  %of CPU
                  ts1: 148743       1225   121        95
                merid: 153062       690    222        98
             moraine: 151495       251    604        97
                  ts2: 148764       1215   122        95
                  ts3: 148727       1219   122        95
                  ts4: 148726       1216   122        95
                  nox: 150528       1306   115        96
                  mar: 147830       476    311        95
                 tarn: 148460       478    311        95
                fjord: 148227       490    303        95
                delta: 150616       495    304        96
              tundra: 150767       468    322        96
              corrie: 145984       299    488        93
```

Table 1 : Extract from final monitor report

The variation in performance between the processors is clear: the pattern was consistent in all runs. The efficiency of use of each processor demonstrates the effectiveness of the task farm.

Figure 10 shows the total number of tasks completed in each hour during a typical run, data written hourly to a different statistics file for each day. These runs were performed during vacation time, so the variation is not very great. There is improvement in efficiency after 5pm; the overnight dip is likely to be a consequence of backup. In two cases, higher statistics are followed by low counts, due simply to the timing with which jobs completed. (10 processors were active in this case. Chance will lead to tasks closing just after the statistics are gathered on some occasions.) The task farm is sufficiently flexible that variation in run-times can be accommodated as other machine use varies; running each task at low priority (by use of the UNIX command *nice*) ensures that other users are not inconvenienced.

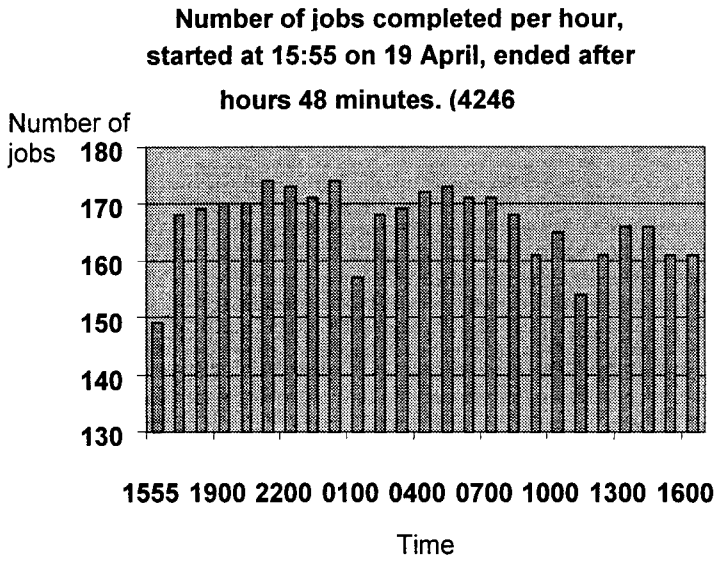


Figure 10 : Number of jobs run per house by time of day

The coordinator used CPU time totalling 1.1% of the elapsed time, including both the coordination script (94 secs in the elapsed 2607 minutes, including the polling for requests), and the child processes running remote shells for each worker task (1690 secs). (The latter figure could be further reduced, if desired, by using threads rather than forked processes.)

Table 2 gives summary performance figures and also extrapolates these for larger data sets.

Rows	Columns	Directories	Files	Raw Size (Gb)	Compressed Size (Gb)	Raw generation time (days)	Sample multi-processing generation time (days)	50 1GHz PC multi-processing generation time (days)	Compressed database read time in java (min)
466	336	117	9828	2.9	0.1	14.0	1.8	0.1	3.9
901	901	226	51076	76.7	2.1	72.8	9.4	0.8	104.4
1201	1201	301	90601	242.2	6.8	129.4	16.7	1.4	329.6
3601	3601	901	811801	19,575.0	547.8	1163.1	149.9	12.2	26634.6

Table 2 : Estimated sizes and generation times of various sizes of DEM

More detailed database access times are reported in the context of the tactical decision aids, discussed next in this report.

5.4 Tool to extract one visibility grid

A tool to extract a single MAP from the database for a specified point has been implemented in Java to reuse the code developed for the tactical decision aids (6.3). The tool is used from the command line and is invoked as follows:

java –cp src uk.ac.ed.geo.cid.ExtractMAP <cid_database_name> <output_file_name> <row> <column>

The additional class to support this tool reads the parameters from the command line, creates an instance of the CID database class, reads the header information to initialise this class and then loads the MAP corresponding to the specified row and column from the database and invokes the write method to create a BIL file.

5.5 Possible developments

The number of files and the small size of the zip files may cause a problem with lost disk space on some platforms. In this case the individual zip files could be collected together in subsets. An extreme case of this would see each row directory replaced by a zip file containing the compressed version of each MAP16 in that row. This format may be more appropriate on CD for example or on FAT16 based Windows systems.

The distribution of the files across multiple disks would facilitate parallel implementations of the tactical aids: indeed the multiprocessing architecture was designed to be extendable to manage multiple types of task with this in mind.

6 Tactical decision aids

6.1 Overview

The commonality between the applications was exploited by use of object-oriented techniques, in Java, as package uk.ac.ed.geo.cid.

Three programs, one for each aid, derive ArcInfo grids in ASCII form or a tabular data of row, column and count in ASCII form. This allows a further program to view either the initial grid or the decision tools' results.

The basic algorithm is described in the following subsections in the form of simple pseudo-code.

6.1.1 Cumulative visibility

Result is a c x r grid of counts (`cumvis`) as shown in Figure 12.

The operation is illustrated in Figure 11 and in the following code:

```
Initialise cumvis to 0.
for each MAP(i,j)
for each row r
  for each col c
    if  (MAP [c, r] == 1)
     cumvis [c, r] = cumvis [c, r] + 1
```

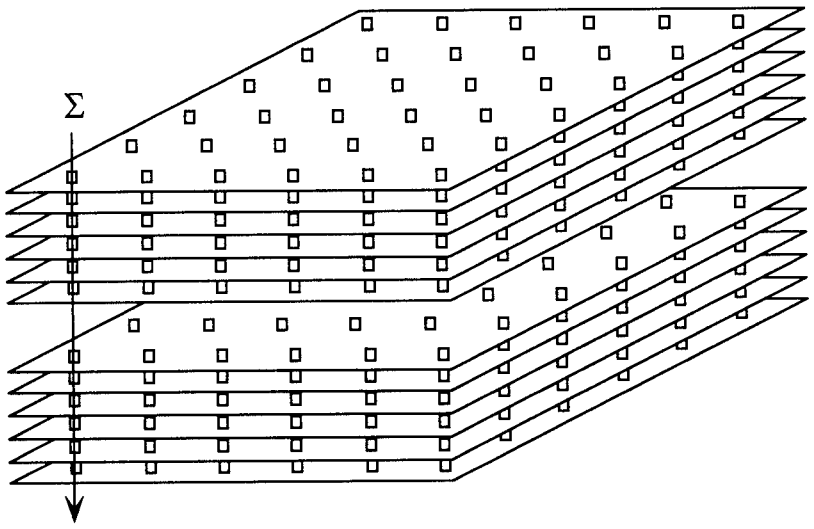


Figure 11 : Cumulative Visibility Operation

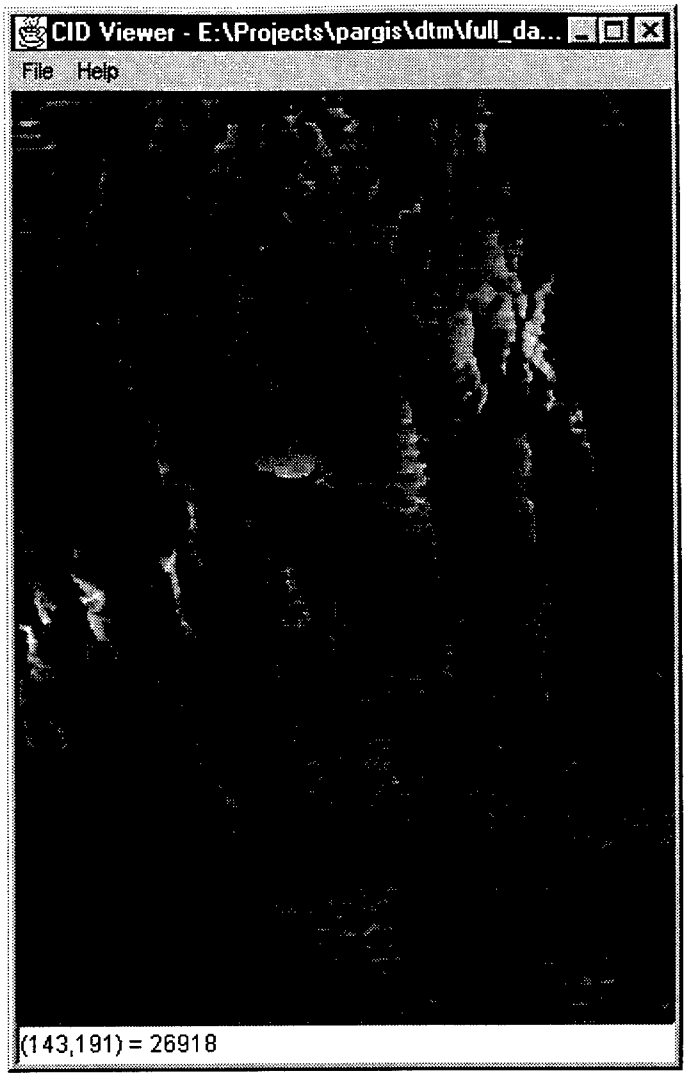


Figure 12 : Display of Results of Cumulative Visibility Operation

6.1.2 Post of maximum visibility

Result is a c x r array of tables each containing a list of `col`, `row` and `count` values as illustrated in Figure 13.

This structure allows the case where more than one post has the same level of visibility from the specified post. The operation is illustrated in Figure 14 and in the following pseudo-code.

For		MVP is		
Row	Col	Row	Col	Visibility
65	23	129	268	57932
65	24	191	138	55445
65	25	42	45	21492
65	26	15	6	24838
65	27	15	6	24838
65	28	15	6	24838
65	29	15	6	24838
65	30	15	6	24838
65	31	15	6	24838
65	32	15	6	24838

Figure 13 : Sample from results of a Maximum Visibility Operation

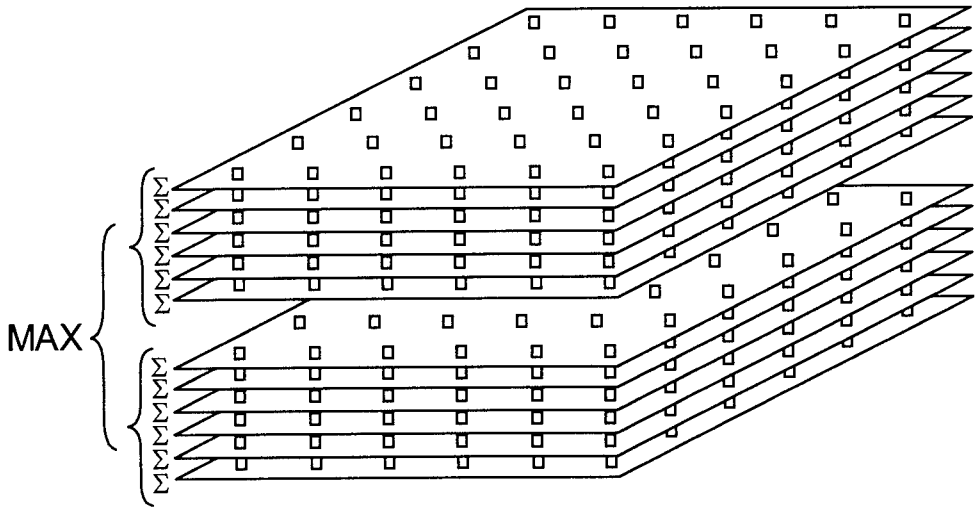


Figure 14 : Maximum Visibility Operation

```
Initialise array of tables to null.
For each MAP(i,j) do
Calculate total visibility of MAP
sum = 0;
for each row l
  for each col k
    if (MAP [k, l]) == 1
      sum = sum + 1

for each row l
  for each col k
    if MAP [k, l] == 1  /* to test is post is visible in this
MAP */
      if sum > count [k, l /* does this MAP have higher
visibility */
      {
         add (i, j, sum) to table of values for cell [k, l]
      }
      else if (sum == count[k, l])
      {
         initialise the table for cell [k, l] to (i, j, sum)
      }
```

6.1.3 Multi-Observer Masked Area Plot

Result is grid of counts, c as illustrated in Figure 15.

Input is a grid representation of the masking polygon (binary 1 = inside).

The operation is illustrated in Figure 16 and in the following pseudo-code:

```
For each MAP(i,j)  do
sum = 0;
for each row l
  for each col k
    if (MAP [k, l] == 1  && poly p[k, l] == 1)
                        /* post is visible and within polygon */
     sum = sum + 1
 c[i, j] = sum
```

6.2 Implementation

These algorithms have been implemented in Java with a base class implementing a general representation and a sub-class implementing versions of the algorithms optimised for the MAP16 representations. All 16 MAPs can normally be processed in one pass providing some optimisation in performance. The code also bypasses processing for all posts where all 16 MAP values are zero. Java provides access to zip files as part of a standard package which simplifies implementation of the algorithms. Performance is reasonable with the operations completing in around 2 minutes on the 41% CID leading to an expected 5 minute completion for a full database in single threaded mode.

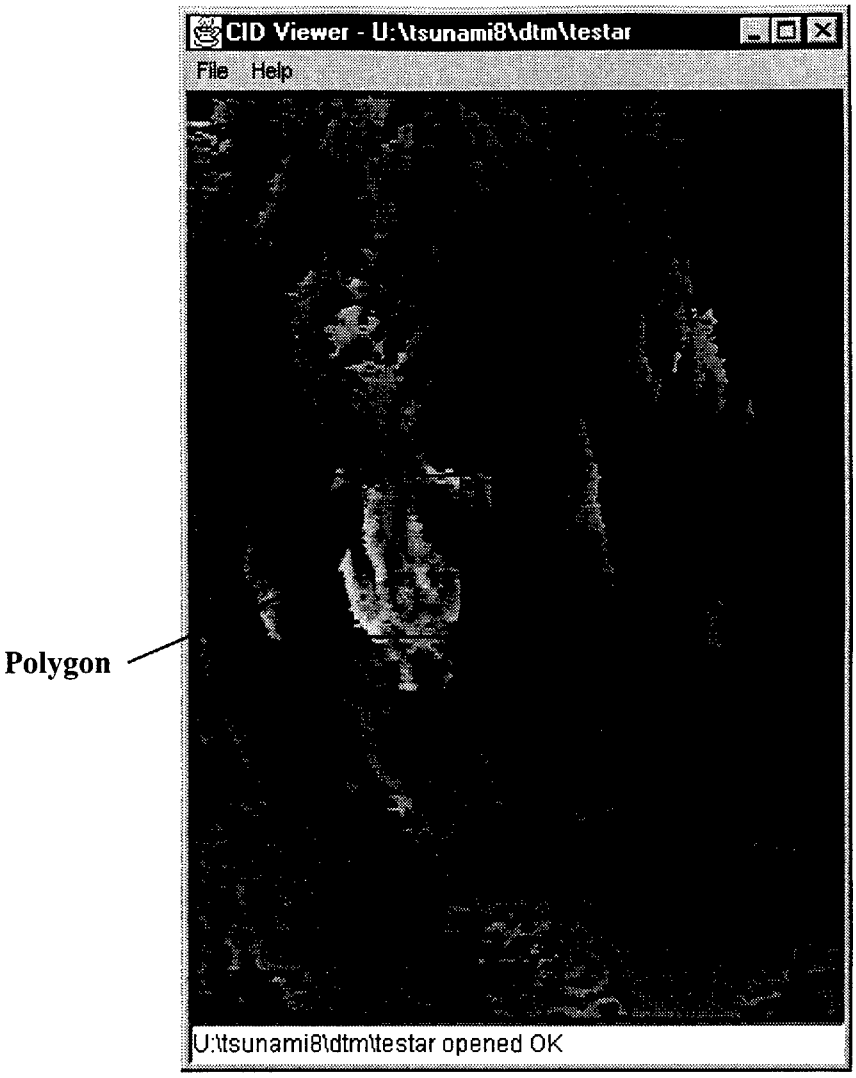


Figure 15 : Display of results of Multi-Observer Masked Area Plot

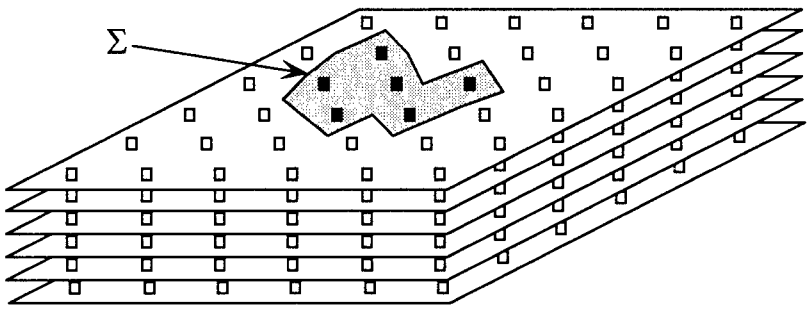


Figure 16 : Multi-Observer Masked Area Plot Operation

6.3 *Software design*

Documentation of the software is provided in standard JavaDoc format in an associated document. An outline description is provided here. The classes developed can be considered in 4 groups:

6.3.1 Classes to represent a single or group of MAPs

A generic class `MAPBase` contains fields and methods to store a set of MAPs as described in 5.2 The internal representation of a set of MAPs is as an array of bytes with a group of consecutive elements in the array representing the MAPs for a rectangular block of posts in the DEM. Static class fields `bitsPerCell` and `bytesPerCell` define the number of bits (and therefore posts) in the group and the number of bytes required to store the group. The default values are 16 and 2 respectively. The class also contains methods to read and write MAPs in ASCII text, binary (BIL) and compressed binary (zip) format. Methods to support calculation of the tactical aids for a set of MAPs are also provided. To allow graphical representation of a MAP on a display screen, the class includes a method for generating an ImageSource. Class `PadString` is provided to allow right-justified numerical output for BIL headers

Class `MAP16` provides some optimisations for the case where a block of 4x4 MAPs are stored together, as described above in 5.2. In particular there are optimised methods to support the three tactical aids described in the project specification. The optimisation takes advantage of the internal representation as a simple array of bytes and processes the data with a single loop, avoiding the overheads of nested loops. The manipulation of single bits is carried out using a the & (and) operator where necessary although the cumulative visibility calculation uses an array to convert the particular bit pattern in a byte into the number of bits set in that pattern as a short cut.

Class `MAP1` is the simple case of a single MAP although in this case we use on byte to store a cell rather than using a single bit. This is for ease of coding since the memory overhead was considered unimportant in this case.

Class `MAPInt` is a specialisation of *MAPBase* which is intended to store a DEM rather than a MAP but it takes advantage of some of the methods provided in the base class.

6.3.2 Classes to represent a CID (complete intervisibility database)

A generic class `CIDBase` contains fields and methods to manage access to a complete intervisibility database stored on disk and the allow operations to be performed on the complete database as well as tools to extract particular MAPs from the database. The methods `preProcessMAP`, `processAllMAP` and `postProcessMAP` allow collective operations to be carried out on the database. The methods `processAllMap` steps through all MAP16s in the database and calls `processMAP` on each in turn. Method `processMAP` is a dummy method intended to be overridden in subclasses implementing particular algorithms.

Class `CIDAreaVis` overrides `processMAP` to call the `polygonMaskedAreaPlot` method on a MAP.

Class `CIDCumVis` overrides `processMAP` to call the `CumulativeVisibility` method on a MAP.

Class `CIDMaxVis` overrides `processMAP` to call the `MaximumVisibility` method on a MAP.

Class `CountXY` provides the fields necessary to store the table of row, column and count values at each post in a maximum visibility analysis and is used in `CIDMaxVis`.

6.3.3 Classes to allow display of a CID

A generic class `DisplayImage` provides a method to allow a CID, a MAP or the result of a tactical aid operation to be displayed. It provides a basic drawing method and a method for handling mouse events. The default response to a mouse click is to display the mouse position and the value of the displayed image under the mouse. Class `CIDCanvas` is used as the graphical component on which to draw the CID and this provides the link between a mouse click by the user and the DisplayImage subclass which is providing the image. `CIDCanvas` also provides methods for updating the status bar in a GUI application.

Class `CIDImage` refines `DisplayImage` to provide support for displaying a grey-scale image of a DEM and for overlaying that with a red/green partially transparent mask showing the areas which are visible in a MAP. The mouse-event handler is written to retrieve the MAP corresponding to the post under the mouse.

Class `IntDisplayImage` extends `DisplayImage` to provide support for displaying a grey-scale image derived from a grid of integer values. This could be a DEM containing height values or a integer grid containing the result of a Cumulative Visibility or Polygon Masked Area Plot operation.

Class `MAPDisplayImage` extends `DisplayImage` to provide to allow display of a set of MAP values.

6.3.4 Applications classes to support command line utilities.

Class `TestCIDAreaVis` reads parameters from the command line and creates an instance of `CIDAreaVis` which is used to read and process a Multi-Observer Masked Area Plot operation. The result of the operation is written out as a grid in ASCII format. The polygon to be used as the mask is specified in the <polygon_name> parameter and is expected to be a grid of the same size as the CID and in ArcInfo ASCII format. The application is invoked from the command line as follows:

java –cp src uk.ac.ed.geo.cid.TestCIDAreaVis <cid_database_name> <polygon_name> <output_file_name>

Class `TestCIDCumVis` reads parameters from the command line and creates an instance of `CIDCumVis` which is used to read and process a Cumulative Visibility operation. The result of the operation is written out as a grid in ASCII format. The application is invoked from the command line as follows:

java –cp src uk.ac.ed.geo.cid.TestCIDCumVis <cid_database_name> <output_file_name>

Class `TestCIDMaxVis` reads parameters from the command line and creates an instance of `CIDMaxVis` which is used to read and process a Maximum Visibility operation. The result of the operation is written out as a table of 5 values with one line per row in ASCII format. The application is invoked from the command line as follows:

java –cp src uk.ac.ed.geo.cid.TestCIDMaxVis <cid_database_name> <output_file_name>

Class `ExtractMAP` reads parameters from the command line and creates an instance of `CIDBase` which is used to read a database and extract the MAP for the post at a particular row and column. The result of the operation is written out as a grid in BIL format. The application is invoked from the command line as follows:

java –cp src uk.ac.ed.geo.cid.ExtractMAP <cid_database_name> <output_file_name> <row> <column>

6.3.5 Application for viewing a CID

Class `CIDViewer` allows a CID and its components to be display in a simple GUI context. It utilitises the widgets defined in class `CIDViewerFrame` to allow the user to select the CID to be displayed, to interrogate the database for the values at particular posts and to select a MAP for display. It may be invoked with a single parameter to specify the name of the database to be displayed. If this parameter is missing then a blank image is displayed until the user selects a valid CID or CID component. Invocation from the command line is via:

java –cp src uk.ac.ed.geo.cid.CIDViewer <cid_database_name>

Figure 17 shows the sample DEM displayed in CIDViewer as a gray-scale image while Figure 18 shows a Masked Area Plot superimposed on the DEM. The arrow indicates the observation post with areas which are visible displayed in green and areas which are not visible shown in red.

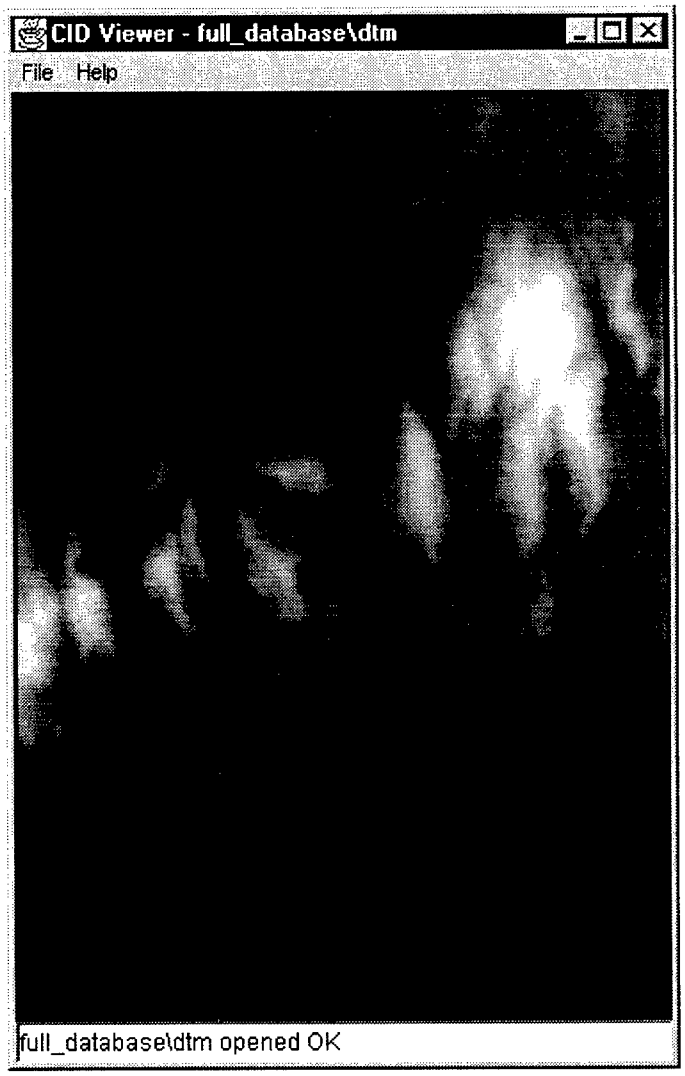


Figure 17 : Display of Digital Elevation Model in CIDViewer

6.4 *Performance*

All timings in this section are based on a Sun Enterprise E450 with 4 x 300MHz processors and 512Mb main memory.

The basic java implementation of tools to access a CID allow reasonable real-time performance for opening and database and retrieving single MAPs. Opening a database takes less than 1 second while retrieving a particular MAP takes around 0.25 seconds. The internal method for reading a MAP16 in zip format take approx 25ms – reading all MAP16s in a database takes approx 233 seconds. This is the base time for processing any operation which requires access to all MAP16s in the database. The processing time for the individual operations is given in Table 3.

Operation	**Time for 336 x 466 database**
Read entire database	233s
Cumulative Visibility	356s
Polygon Masked Area Plot	338s
Maximum Visibility	706s

Table 3 : Processing time for individual operations

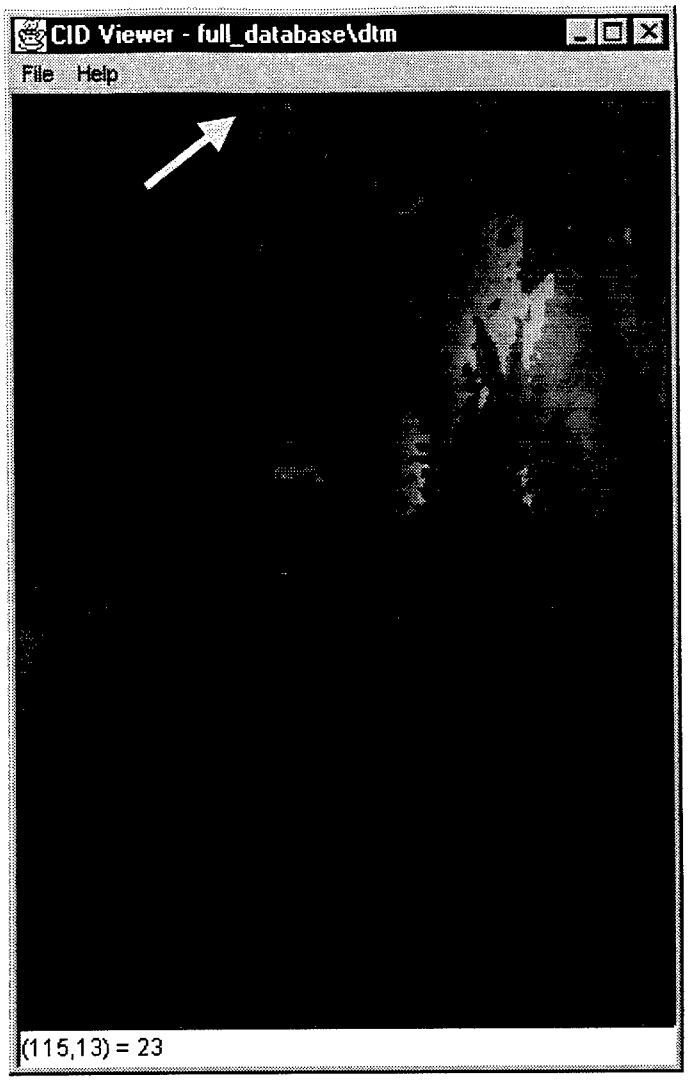


Figure 18 : Display of DEM and a Masked Area Plot

6.5 Examples of results

Examples of the results from the tactical aid operations have been shown above in sections 6.1.1, 6.1.2 and 6.1.3. The first and last examples were displayed using CIDViewer. The results derived from the sample data show some obvious artifacts which are the result of ripples in the DEM data in the North-South axis.

6.6 Possible developments

The design of the software is sufficiently general that it could be adapted to different methods of compression and database design could be added relatively easily. Since the database design is file based and the software is implemented in Java it would also be possible to serve the database using a standard web server and a slightly modified viewer program running as an applet in a browser. Some simple packaging of the viewer would also allow conformance with OpenGIS standards for web browsing and therefore use of the intervisibility database with other servers.

7 Future directions

The framework described in this report is very much a prototype, although the technology has been proven to give a significant performance advantage. The prototype could be developed in the future in a number of directions, including:

1) Add functionality to the task farm. Possibilities might include:
   a) Allow multiple types of task to be co-ordinated in the task farm
   b) Increase the flexibility of scheduling tasks
   c) Allow the ArcInfo command to be generated by the task farm, reducing the processing in each worker but increasing the range of possible analyses. The merit of this approach is in cases where the choice of analysis might be subject to previous results.
   d) Extend the task farm to support additional packages and applications
   e) Extend to support mixed operating systems
   f) Extend to support a secure operating environment
   g) Running more CPU intensive tactical decision aids in parallel, perhaps with a sink process to collate results from those aids.

2) Extend flexibility of the database
   a) Permit the database to be distributed across several physical disk drives
   b) Web-enable the database, taking advantage of the efficient compression already incorporated in the prototype.
   c) Store the inverse CID with aggregate visibility measures

3) Add additional tactical aids, for example
   a) multi-post visibility analysis
   b) continuous/contiguous visibility analysis

Appendix: Multicomputing Architecture Software

7.1 Directory structure

- Tec top directory, on a shared disk, accessible to all workers.
  - Tec/tasks top directory for tasks. (We are most of the way to having a task farm that can manage multiple tasks with ability to switch between different tasks during a run, or partitioning workers to run two task farms concurrently.)
    - Tec/tasks/cit Complete Intervisibility Database files
      - Tec/tasks/cit/cit_database Gathers data from all processors.
      - Tec/tasks/cit/run Template files for a run: these are copied to processors' run directories. All scripts and programs specific to this CIT task are held here
      - Tec/tasks/cit/inputdata Holds original ArcInfo data.
    - Tec/coord Task farm coordination files: generic to all uses of the PUFFIN framework. Major controlling files (workers.names, controlTaskFarm.perl are here.)
      - Tec/coord/wksetup Scripts copied to workers' run directories
      - Tec/coord/wkreq Receives files constituting requests for tasks from workers. Also holds a file of the last task delegated to a worker. One request and one sent file per worker are held.
      - Tec/coord/taskFailed Any trace files from tasks recognised to have failed are copied into here by workers.
      - Tec/coord/taskDone Each completed task has a file in here, generated by the workers, read by the coordinator to generate profile of activity, and by the monitor utility.
      - Tec/coord/monitor Profile and monitoring data reside here.
- Worker directory Usually a sub-directory of /tmp. Defined in workers.names (Figure 19)
  - ./inputdata Copy of tec/cit/inputdata taken once at initialisation
  - /run files copied from Tec/coord/wksetup and Tec/tasks/cit/run

  THESE FILES SHOULD NEVER BE EDITED DIRECTLY. They are overwritten at intiation of the task farm, by files from the tec/coord/... and tec/cit.... directories.

7.2 Installation

One tar file, tec.tar contains the directory tree tec/...

It should be written to a filesystem with access from all workers. The current system assumes that the files have the same directory name in all processors and the same username/account is registered on each machine.

Workers are configured as follows:

- Are defined in *.rhosts* on the coordinating processor to allow the coordinating processor to run rsh commands. (Could be replaced by ssh in future)

- Are registered in *workers.names* (Figure 19)
- Require PERL and ArcInfo to be available
- Require access to the shared tec directory tree
- Do not require any user intervention during a task farm run, normally.

A new DTM is made available by use of an installation scripts, install.perl and installdata.aml. These create an ARC GRID, and also a points database, comprising the coordinates of each DEM point.

7.3 Invocation of ArcInfo from PERL

In this description we work backwards, from the mechanism used to run commands in ArcInfo to the PERL scripts. The invocation of ArcInfo from a script is typically accomplished by use of the ArcInfo scripting language AML. The UNIX machines all used bash as a default shell, with environment parameters set up from system-wide bash scripts. Therefore the AML command was invoked from a bash script. The parameters for this bash script were determined by the PERL script.

For example, to install data, the PERL includes:

```
$sourceGrid=$ARGV[0];
$sourceDir=$ARGV[1];
system("runArc \"installdata.aml $sourceGrid $sourceDir
../inputdata\"");
```

and the runArc bash shell script is generic:

```
#!/usr/local/bin/bash
amletc=$1
#echo "run with " $amletc
. /etc/profile > /dev/null
$ARCHOME/bin/arc \&run $amletc
```

Finally, the installdata.aml copies a pre-existing GRID file, removes any null's from the grid, and creates a point database for each cell, with height associated with each point:

```
&args grid gridin citinput

&type move to working directory citinput
&workspace %citinput%
&type  use grid_vis for the visibility DEM.
copy %gridin%/%grid%  grid_vis
&type removing nulls
gridimage grid_vis NONE fred_im bil
```

```
imagegrid fred_im grid_in

&type making point database used to select viewpoints
gridpoint grid_in points_in HEIGHT
addxy points_in

&type also hold an ASCII grid needed by the decision aid tools
gridascii grid_vis dtm
kill grid_in
&sys rm -R fred_im*
&return
```

7.4 Invocation of the Visibility analysis

This follows the pattern described above: PERL calls bash which calls ARC with an AML script and parameters. It is executed locally on each worker, using local disks:

- Header file: gives size of the grid; created by install script.
- Inputdata directory with point and grid data, set up by the same install script.
- The task number passed by the coordinator task farm
- A simple algorithm to derives a bounding box that contains 16 points, and extracts these points from the point-database into a temporary database. It also derives the block row and column.

The processing is thus:

1) From the task number and the file header, derive block bounding box and top-left coordinates of grid block.
2) Invoke runArc with AML script arcvis.aml which:
   a) select points within the block bounding box from the point database
   b) run the visibility analysis
   c) convert the resulting grid to a BIL image
   d) compress the image using zip
   e) delete temporary files
3) Scan the trace file from this task for error statements. If one is found, copy the tracefile to the coordinators taskFailed directory, as *task_number.trace.*

(The error trap was activated once only, when an error in generating the block bounding box resulted in no points being chosen. That error was fixed.)

The script arcvis.aml is:

```
&args grid points outputworkspace tempworkspace xmin xmax ymin
ymax row col
```

```
&type Shared results directory is %outputworkspace%
&type Own working directory is %tempworkspace%
&workspace %tempworkspace%
&sv outname = %row%
&sv tempname = %row%_%col%

&if [exists %outputworkspace%/%row%/%col%_im.zip -file] &then
     &return

&sv tempp := %tempname%_P
&type Reselecting from (%xmin%,%ymin%) to (%xmax%,%ymax%)
reselect %points% %tempp% point
reselect X-COORD >= %xmin% and X-COORD < %xmax%
     and Y-COORD >= %ymin% and Y-COORD < %ymax%
~
NO
NO

visibility %grid% %tempp% point %tempname% GRID observers
gridimage %tempname% NONE %outname%_im BIL
&sys /usr/local/bin/zip %outputworkspace%/%row%/%col%_im.zip
%outname%_im.bil

kill %tempp%
kill %tempname%
&sys rm %outname%_im.*
&return
```

7.5 The monitor utility

The script monitor.perl, run form the coordinator's directory, lists information related to jobs completed and in progress. If any of the active processors have yet to complete a task, then the message "processor name – check configuration. Done Nothing" is written out. This is always the case at the start of a run, when no error is present. Figure 20 shows an example of output from running monitor.perl.

A command-line option alerts the user if one task has been active for more than twice its average run-time. The option is invoked by adding any text to the command, e.g. "monitor.perl check". If a slow task is recognised then the processes running on the worker are checked to see

is workerTask is among those owned by the user. If so, the message "processor name seems to be still running workerTask" is written out. If not, then the user is asked if a restart is required.

If a processor is known to have been closed down, and is restarted, then the command "restartWorker.perl workername" will reconnect the worker to the task farm assuming the workers directories were initialised correctly before the close-down.

The monitor script lists all gaps in the database: in the event that no valid observation points exist (the DEM is undefined in the region, as happens in the test data at some parts of its margin) ArcInfo considers an error to have occurred. The trace file of the task is written to the coordinator's taskFailed directory, but the monitor utility counts both the total number of failed tasks, and the number to have "failed" due to the lack of observation posts. In the 9828-task run with the test data, 53 tasks terminated with no observation posts, and no other errors occurred.

7.6 Task Farm Coordinator script

Figures 22 and 23 show examples of the output from the coordinator script during initialisation. Figure 24 shows an example of the output from the coordinator following invocation of the *causeStop* script.

7.7 Other script utilities

Additional utilities have been written to:

- To scan database for the first missing task (used on closure of the task farm, so the next task farm session begins at this task)
- To list all gaps in the database and the last task to have completed.
- To restart a processor in the task farm
- To check activity on all workers

7.8 State of the code

The code functions as a prototype. It may be desirable to improve the following:

1) The overhead of the task farm management. This is not high at present. With 11 processors, on an occasion when the CPU use on the visibility jobs was above 70%, the task farm management itself costs under 1% of a CPU. The processor running the coordinator process also completes the visibility jobs with an efficiency in excess of 90%, with other unrelated system activity continuing. The wait-time within each processor has not been measured. As additional processors are added (none were available here), then it might be desirable to:

   a) Increase the number of jobs run with each task. Current run-times are in the order of 2 to 5 minutes. Thus is would be reasonable to run 5 or 10 jobs in each task. If urgency was essential then the number of job per task would be reduced as the end of the complete dataset generation was approached.

   b) The mechanisms employ the use of multiple small disk files, one for each completed task. The task farm coordinator reads each once only in generating its hourly profile of use; it then renames each. The monitor utility in particular currently reads all these files (extensions *done* and *DONE*). It would be better to read and delete these from the monitor utility, holding a summary between monitor runs.

2) Clean up /tmp directories. This is simple to accomplish but is not included. It was preferred to leave trace files in the workers' run directories to allow checks of any anomalous results.

3) Portability to NT. Achieving this was beyond the intention and scope of the project. The *workers.names* directory includes the operating system of each worker named. To run on NT, or a combination of operating systems, some perl scripts will need to be amended either:

   a) Branching according to the operating system of the worker

   b) Invoking separate scripts (with NT system calls, and NT-style directory names, for example)

   The extent of these changes has not been assessed.

```
11
ts1 tsunami unix /tmp/ts1 /usr/local/bin
ts2 tsunami unix /tmp/ts2 /usr/local/bin
ts3 tsunami unix /tmp/ts3 /usr/local/bin
ts4 tsunami unix /tmp/ts4 /usr/local/bin
nox nox unix /tmp/mjm /usr/local/bin
mar marsh unix /tmp/mjm/ /usr/local/bin
tarn tarn unix /tmp/mjm/ /usr/local/bin
fjord fjord unix /tmp/mjm/ /usr/local/bin
delta delta unix /tmp/mjm/ /usr/local/bin
tundra tundra unix /tmp/mjm/ /usr/local/bin
corrie corrie unix /tmp/mjm/ /usr/local/bin

# This file should exist in the coordinator task directory.
# line 1: no. of workers to use.
# lines 2-(1+no. of workers to use)
# unique_name IPname opsys tempDir perlDir
# where:
#  unique_name: allows multiple processes per IP name, as with an SMP machine
#  IPname: hostname for rcp to worker, and rsh to invoke a run on worker.
#  opsys: not used... assuming UNIX filesystems and some UNIX commands.
#  tempDir:  directory used for local temporary storage.
#  perlDir: where to find perl.
```

Figure 19 : workers.names file

```
Started this session at task 3392, 25 hr 48 min ago
 Number of workers active: 10
```

Active tasks section removed from table: workers wait for closure of each, so the final figures here are not meaningful: these are used during a run, not after it.

```
 Total CPU time used on 4246 tasks: 862466 (=elapsed * 9.28)
 Last task required will be 9828

 Batting averages:  CPU total       ntasks  CPU/task  %of CPU
           ts3:        83971         669     126        90
           ts2:        84219         669     126        91
           ts1:        84006         669     126        90
           ts4:        84018         669     126        90
           mar:        87371         280     312        94
           tarn:       88026         280     314        95
           fjord:      88168         283     312        95
           delta:      88463         279     317        95
           tundra:     88720         278     319        95
           corrie:     85505         170     503        92

 Failed Tasks:
 Total number of failed tasks: 0
 First gap in database is for 7639
```

Figure 20 : Output from monitor.perl

```
Started this session at task 321, 0 hr 17 min ago
 Number of workers active: 11

 Active tasks: elapsed time, task, worker
        471 secs: 355 on corrie
        17 secs: 377 on delta
        32 secs: 376 on fjord
        300 secs: 365 on mar
        118 secs: 371 on nox
        335 secs: 363 on tarn
        67 secs: 373 on ts1
        67 secs: 374 on ts2
        62 secs: 375 on ts3
        72 secs: 372 on ts4
        315 secs: 364 on tundra

 Total CPU time used on 46 tasks: 8479 (=elapsed * 8.30)
 Last task required will be 9828

 Batting averages:  CPU total  ntasks  CPU/task  %of CPU
               ts1:     755       6        126     74
               ts2:     755       6        126     74
               ts3:     757       6        126     74
               ts4:     757       6        126     74
               nox:     850       7        121     83
               mar:     663       3        221     65
               tarn:    946       3        315     93
               fjord:   924       3        308     90
               delta:   940       3        313     92
               tundra:  627       2        314     61
               corrie:  504       1        504     49


Checking taskFailed files for failures recognised by workers

 Total number of recognisably failed tasks: 0

First gap in database is for task 327
```

Figure 21 : Example of monitor output

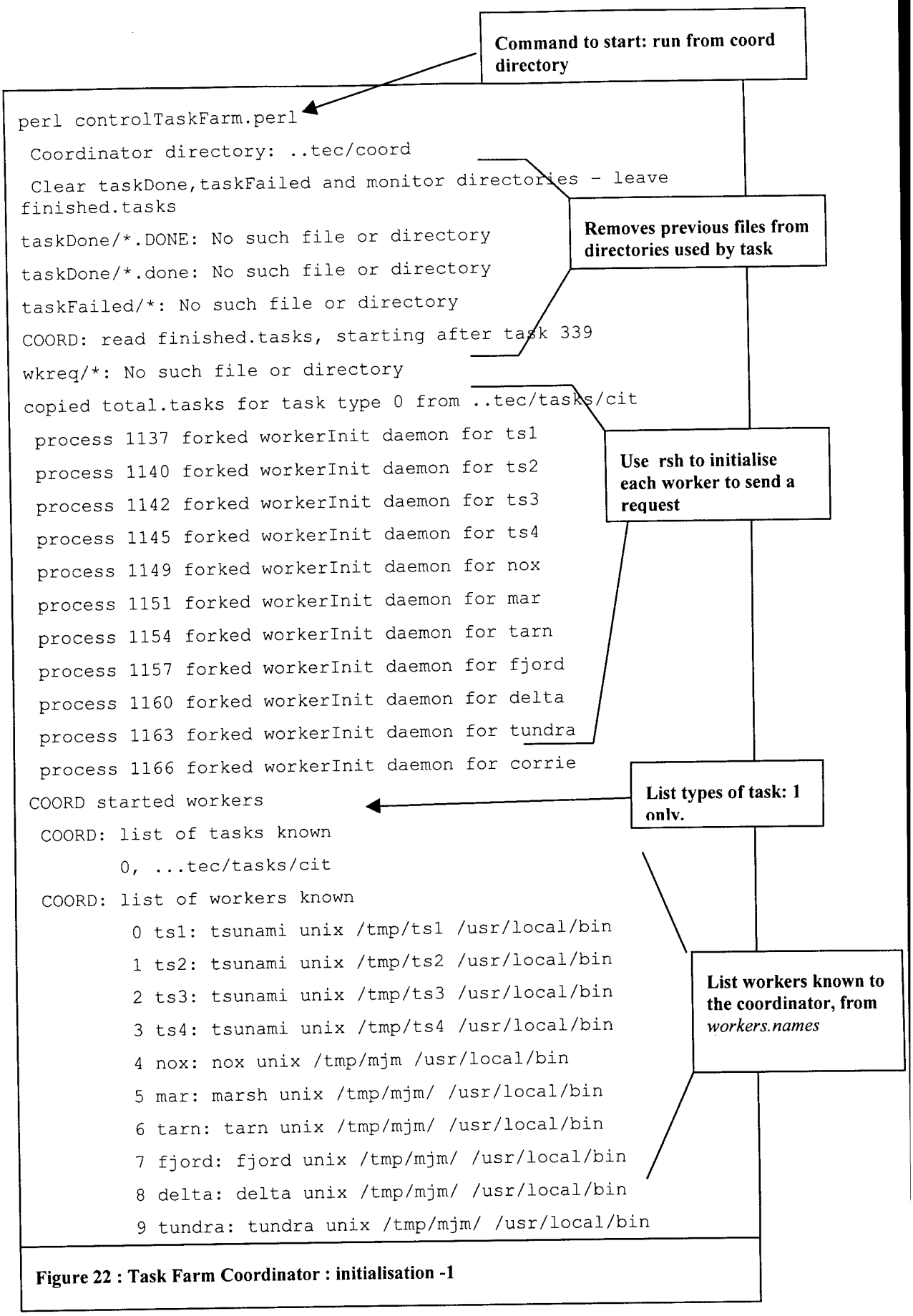


Figure 22 : Task Farm Coordinator : initialisation -1

```
9 tundra: tundra unix /tmp/mjm/ /usr/local/bin
10 corrie: corrie unix /tmp/mjm/ /usr/local/bin

mkdir: Failed to make directory "/tmp/ts2"; File exists
mkdir: Failed to make directory "/tmp/ts4"; File exists
mkdir: Failed to make directory "/tmp/mjm"; File exists
mkdir: Failed to make directory "/tmp/ts1"; File exists
mkdir: Failed to make directory "/tmp/ts3"; File exists
mkdir: cannot create /tmp/mjm/.
/tmp/mjm/: File exists
mkdir: cannot create /tmp/mjm/.
/tmp/mjm/: File exists
mkdir: cannot create /tmp/mjm/.
/tmp/mjm/: File exists
mkdir: cannot create /tmp/mjm/.
/tmp/mjm/: File exists
mkdir: cannot create /tmp/mjm/.
/tmp/mjm/: File exists
mkdir: cannot create /tmp/mjm/.
/tmp/mjm/: File exists
workerInit on ts2 finished
workerInit on ts4 finished
 workerInit on ts3 finished
 workerInit on ts1 finished
workerInit on nox finished
Process 1297 forked daemon to send task 340 to nox
 Process 1302 forked daemon to send task 341 to ts1
 Process 1307 forked daemon to send task 342 to ts2
 Process 1312 forked daemon to send task 343 to ts3
 Process 1317 forked daemon to send task 344 to ts4
 workerInit on mar finished
 workerInit on delta finished
 workerInit on corrie finished
 Process 1333 forked daemon to send task 345 to corrie
```

Create the worker's top directory if it doesn't exist. Do this by rsh. If it does exist these messages are seen and are not a problem. (An example of untidy-prototype not live-quality functionality.)

Text from worker, when workerInit finishes. workerInit sends a request to the coordinator.

On receipt of a request, the coordinator sends a task to the worker. It does this from a daemon (detached) process, forked from the coordinator.

Figure 23 : Task Farm Coordinator : initialisation - 2

```
various trace according to shell
login/profile scripts in use.
```

Closedown is caused by running causeStop or by completing all tasks. Coordinator counts the workers as they complete last assigned-task.

```
 worker nox finished
 worker ts1 finished Waiting for 9 of 11 workers to finish
 worker ts4 finished Waiting for 8 of 11 workers to finish
 worker ts2 finished Waiting for 7 of 11 workers to finish
 worker ts3 finished Waiting for 6 of 11 workers to finish
 worker delta finished Waiting for 5 of 11 workers to finish
 worker mar finished Waiting for 4 of 11 workers to finish
 worker tarn finished Waiting for 3 of 11 workers to finish
 worker tundra finished Waiting for 2 of 11 workers to finish
 worker fjord finished Waiting for 1 of 11 workers to finish
 worker corrie finished Waiting for 0 of 11 workers to finish
...goodbye
 The first missing task in the database is 405
Coordinator ran for an elapsed time of 554 secs
 Wait time was 97.1119133574007 %
```

Scans CIT database from 1st task to find first missing file

Figure 24 : Task Farm Coordinator : Closedown